Measuring Qualities for OSGi Component-Based Applications

Salma Hamza
IRISA
University of South Brittany
Vannes, France
Email: salma.hamza@irisa.fr

Salah Sadou
IRISA
University of South Brittany
Vannes, France
Email: salah.sadou@irisa.fr

Régis Fleurquin
IRISA
University of South Brittany
Vannes, France
Email: regis.fleurquin@irisa.fr

***Abstract*—Component-based software engineering (CBSE) begins to reach a certain level of maturity. Indeed, for the development of complex applications the use of component paradigm has become common. Therefore, the evaluation of the quality of these applications becomes necessary. In this context, the use of metrics is considered very important. Several metrics specific to component-based applications have been proposed. However, any of these metrics gained the consensus of the CBSE community and mainly there is no proposed tool to support them.**

As a large part of frameworks for component-based application development is based on object-oriented technology, we propose to use some object-oriented (OO) metrics to evaluate component-based applications produced with this kind of framework. Indeed, these metrics became a standard in OO community. So, they are well-defined, well-known and empirically validated. To identify which object-oriented metrics are useful for the evaluation of component-based applications, we have conducted an experimental study on 10 OSGi applications. This study also gives us the opportunity to discuss on the respect by OSGi developers of some properties pointed out by the literature.

***Keywords*-Component properties, OO metrics, OSGi framework.**

I. Introduction

Component-oriented programming is becoming well accepted by the industry. Its adoption by the industrial is due to the expected benefits such as reliability, reusability and maintainability. But to ensure that a component-based application meets these quality characteristics, components and architectures should respect some "good" properties. Many quality models have been proposed to help in defining the quality of components [1], [2], [3]. In the same way, the architectures should adhere to sound software engineering principles. The two principles that are most often sought in the literature are high cohesion and low coupling. Other properties such as the absence of cycles, adequate size and complexity are also often cited.

The quantitative estimation of these properties is a good means for component developers and architects to predict the quality of their produced artifacts. To realize this aim, they need some dedicated metrics. Several (product level) metric or set of metrics have been proposed specifically for the component paradigm [1], [4], [5], [6], [7]. But as stated by [8] in their survey: i) there is no consensus yet on many of the concepts and elements that are measured by these metrics, ii) inconsistencies in component definitions can frequently be found in many studies, iii) they lack a statistically significant experimental validation and a set of experimental data to provide better insight into their use. Last, but not least, these metrics are not supported by code analysis tools yet. Thus, developers and architects do not (and cannot) use these metrics. Without such product level metrics, an accurate planning and control of the development process are more difficult to achieve.

In this paper we propose a pragmatic approach to help developers and architects to measure some of the essential structural properties that affect the quality of their components and component-based applications in the case of the OSGi framework. In the literature, there is some consensus that component metrics require a different approach from that of structural software or object-oriented metrics. We do not deny the need to provide specific answers to some aspects of the component world such as the measure granularity, the black-box vision and the concept of interface. But in the absence of dedicated standards and tools, we advocate that it is possible and appropriate, in the OSGi framework, to rely on some well-known object-oriented metrics. We will show that some of them can provide valuable information both at the component level (for the developer) and at the application level (for the architect). Furthermore, these metrics have the advantage of being well-defined, the subject of numerous empirical validation and so well-understood, and especially to be computable by most of the code analysis tools.

Thus, we have conducted an experimental study on 10 OSGi applications in order to identify the object-oriented metrics that are useful for this component model. Based on this experimentation, we also propose to quantitatively draw what are an actual typical component and application according to the internal properties pointed out by the literature. This gives us the opportunity to discuss on the importance and the respect of some of these properties by OSGi developers.

The remainder of the paper is organized as follows: Section II describes the method used to identify the object-oriented metrics suitable for component-oriented development. In section III we give the results of our experimentation accompanied with some analysis. We discuss the obtained

results in section IV. Section V describes the validity and the limits of our study. Section VI describes the related work followed by a conclusion in section VII.

II. Approach

We want to prove that object-oriented metrics can be useful in the component paradigm. Our demonstration consists of the following steps:

1) Select a component model in which the object-oriented metrics can be calculated using available code analysis tools.
2) Select a representative sample of component-based applications in the chosen component model.
3) Identify object-oriented metrics that may be candidate (meaningful at the component and application level).
4) Apply candidate metrics on the sample.
5) Use statistical methods to determine the significant (subset of) metrics among the candidates.

In the following, we give the applied criteria to select the component model, the sample of component-based applications, we describe how we select the candidate metrics and finally, we detail the used statistical method.

A. Component Model Selection

Because we want to be able to use "as is" the available code analysis tools, it is necessary that we place ourselves in a component model based on the object paradigm. In such component models, according to [9] a component may be considered as a group of classes that collaborate to provide a system functionality. Some of these classes are "public" (exported), so they form the provided interfaces of the component. Some classes are defined elsewhere (imported), so they form the required interfaces of the component.

Among all the existing component models (a survey can be found in [10], [11]), several are based on the object paradigm. To select the appropriate component model, we admit that the part of the code that is specific to the component framework should not impact significantly on measures targeting the code that is specific to the component. To meet these criteria, we decided to choose only applications developed within the OSGi component model. Indeed, in OSGi, except business classes, there are no additional classes to define the component.

In OSGi, a component is known as a bundle. Each bundle is defined by a single JAR file which packages the module (code and resources) and a manifest file which contains the extra metadata. In fact, the logical bundle is equated with the physical bundle JAR file (the module) [12]. The manifest file is used to describe the bundle's modularity characteristics and to specify mainly the exported packages that are meant to be shared and conversely the imported packages that it required.

B. Application Selection

We set the following selection criteria on candidate OSGi applications:

1) Provide an access to their source code. In fact, the used metrics (see below) require access to the source code.
2) Cover different domains to avoid the characteristics related to a given application domain.
3) Different sizes (number of components in an application and number of classes in a component).
4) Belonging to different development teams to avoid the characteristics associated with habits and good practices common to the same team.

Using these criteria we have selected 10 applications. Table I provides a short description of these applications. They are all open-source OSGi projects and developed in JAVA.

Table I: Selected applications

Application	Description	Nb. of components	Code Size (KLOC)	version
MAT	Eclipse Memory Analyzer Tool	13	86	1.2.0
STEM	Spatiotemporal Epidemiological Modeler Tool	15	35	1.4.0
Eclipse E4	Eclipse Platform	25	20	4.0
SCOUT	Framework for modern service oriented business applications	27	207	3.8.0
IMP	IDE Meta-Tooling Platform	33	117	0.2.1
G-Eclipse	Framework for Grid and Cloud Computing	42	168	1.0
Equinox	OSGi-Framework	50	96	3.3
BIRT	Business Intelligence and Reporting Tools Project	93	1003	3.7.0
OSEE	Framework Open System Engineering Environment	98	290	0.9.6
MoDisco	Eclipse-GMT project	115	510	0.10.0

As we can see, the set of selected applications covers a fairly wide spectrum of application domains and has different sizes (from $\sim$ 20 KLOC to >1,000 KLOC and from 13 to 115 components). All these projects provide an access to their source code via a shared CVS-based software repository.

C. Candidate Metrics

Among the several dozen of existing object-oriented metrics, we consider only metrics which consider the class as the minimal granularity level. This is due to our assumption which considers that components are built by reusing mostly some existing classes. Thus, we do not need metrics measuring the internal structure (class size, class cohesion, method complexity, etc.) of a class.

In particular, work on restructuring object-oriented applications into component-oriented applications [13], [14], [9] indirectly suggest some object-oriented metrics related to two main properties:

- Internal cohesion: a good component should include classes that interact with each other to provide a specific set of functionalities. Therefore, the strength of these interactions is called cohesion.
- Coupling: One of the strengths of component-based development is that its components are loosely coupled and can be combined to build applications. Weak coupling promotes maintainability and reusability of components.

Other properties were also suggested to assess the quality of identified components in an object-oriented application (non-belonging to a cycle, semantic cohesion, granularity, etc.) [9]. These properties were initially used in the evaluation of packages but they also can be applied at the component level. We select all the object-oriented metrics related to this properties.

In the following subsections, we present the selected candidate metrics organized according to their scope (component level or application level). Thus, a metric may target the internal organization of a component (for instance its cohesion), its interfaces or its dependencies with other components (coupling). The two first cases concern the component scope while the last case concerns the application scope. Note that the first and second case should be interested by the component developers. The second and third case should be of interest for the architect.

1) Component Internal Metrics: Metrics that we classified as internal are those relating to its size or its internal dependencies.

a) Size Metrics: Metrics concerning size aspects constitute an important checking tool as the size and the organization of a component have an obvious impact on its comprehension and thus on its maintainability [15]. The selected metrics for component size aspects are summarized in the table II.

Table II: Component's size Metrics

Abbr.	Definition
NP	Total number of contained packages.
Na	Total number of contained abstract classes or interfaces.
Nc	Total number of contained classes.

In this category of metrics we might have integrated the LOC (total number of lines of code) metric. But as our granularity is the class, the Nc metric covers this aspect of size. All these metrics are basic measures that several existing tools allow us to measure at the component granularity.

b) Internal Dependency Metrics: The internal dependency metrics are those that inform the relationship between the component's internal elements.

According to [16], local dependencies can have a global effect on testability of the system. The testability of a component has a direct impact on its maintainability. For component granularity, local dependencies means component internal dependencies. Table III summarizes the object-oriented internal dependency metrics that we believe are relevant to the component paradigm.

We used the LTD metric to measure interaction between classes within a component and the ACD metric, proposed by Lakos [17], to measure the dependency between these classes. Martin's cyclic package metric (PDC) was proposed for object-oriented design [18]. It intends to verify that there is no cycle between packages. In fact, packages related with cycles are more difficult to maintain. We estimate that this metric is interesting for software component, especially in the case where exported packages (case of OSGi) are contained in the cycles. Relational cohesion metric (RC), defined by Larman [19], measures how strongly the elements in object design are related. In our case, elements are classes contained in the component. Thus, RC gives the average dependency of a component's classes. They should be strongly related to get a high cohesion of their component.

Table III: Component's Internal Dependency Metrics

Abbr.	Definition
LTD	Total number of type dependencies (in and out) between all contained classes.
PDC	Total number of cyclically coupled packages contained in a component.
ACD	Average of the number of contained classes that each contained class directly and indirectly depends on. $$ACD = \frac{1}{Nc}\sum_{i=1}^{Nc} CD_i \quad (1)$$ **with:** CD_i = the number of classes that the class i depends on directly and indirectly. Nc = Number of classes in the component
RC	the cohesion between the component's classes. $$RC = \frac{LTD}{Nc} \quad (2)$$

2) Component Interface Metrics: Component interface metrics are those which concern only the visible part of the component. Table IV summarizes the metrics from object-oriented paradigm that we identified as related to component interfaces.

Table IV: Interface Metrics

Abbr.	Definition
ExpP	Total number of exported packages.
Abs	Abstractness only for exported packages. $$Abs = \frac{1}{n}\sum_{i=1}^{n} \frac{Na(i)}{Nc(i)} \quad (3)$$ **with:** n = Number of exported packages in the component

In the case of OSGi applications, exported packages are explicitly defined in the component's *Manifest* file. This allows us to check the total classes of the component to be shared with other components.

The *Abstractness* (Abs) metric proposed by Martin [18] for object-oriented software evaluation will be applied only on exported packages. This metric gives an information on the quality of a component's interfaces. Indeed, a well designed component is supposed to export only interfaces (in Java meaning) or at least abstract classes.

3) Application Metrics: Application level metrics are those that mainly concern the relationship between the components of an application. In other words, these metrics measure the external dependencies of components in the context of their application. According to [20], dependencies reflect the

potential for a component to affect or to be affected by the other components from the same system.

In table V, we list the metrics from object-oriented paradigm whose scope has been adapted to the component level in order to measure its external dependencies.

Table V: Application Metrics

Abbr.	Definition
RTD	Total number of type dependencies (in and out).
CA	Afferent Coupling gives the total number of components that use the measured component (Incoming dependencies).
CE	Efferent Coupling gives the total number of components that are used by the measured component (Outgoing dependencies).
Dep	Number of components from which the component under discussion directly and indirectly (transitively) depends upon (including itself).

The two metrics `CA` and `CE` measure respectively incoming and outgoing dependencies between components. They allow us to measure the coupling degree between components of the same application. The coupling degree is the most important characteristic in the context of reuse [21]. Indeed, a high coupling between the components implies that any change on one of them may affects many others. The `Dep` metric is an object-oriented metric proposed by Lakos [17]. We have adapted it to the component context in order to highlight the transitive dependencies between components of a same application.

The `RTD` Metric gives the number of dependencies (in and out) of classes located inside a component with classes that are outside it (in other components). In the case of OSGi, only classes belonging to packages that are exported or imported are concerned. Thus, this metric measures the level of interaction of a component in an application.

D. Used Statistical Method

We use the descriptive statistics to study the distribution of each candidate metric on each selected application. This statistical method provides simple summaries about the sample (set of components for each application) and the measures (candidate metrics). As we study the metrics individually, we use the univariate analysis.

Thus, our study involves the following steps:

1) We measure the central tendency of the components. In statistics, the term "central tendency" relates to the way in which quantitative data tend to be distributed around some values [22]. In our case, it reflects the tendency of the components and allows us to determine a "typical" or a "central" component around which the other components tend to congregate. The two traditional measures of central tendency, that we used, are *mean* as arithmetic average and *median* as middle score.
2) We use standard deviation which is one of the most frequent measures of spread. It is useful to refer to the variation to validate the emerged results.
3) We analyze the distribution's shape of each variable (metric). For this aim, we first use *Skewness* which is a measure of the asymmetry of the distribution. It indicates whether the data is symmetrical or not around the mean and describes the dispensing side with the high frequency. If data is symmetrical (skewness around 0) then we calculate normalized kurtosis to measure the degree of peakedness of the distribution. In our case, the peakedness property means that there is an excess frequency of components around the central component.

Finally, to illustrate our data's distribution we use BoxPlot representation. It is an understandable and useful graphically way to summarize a collection of data. It is a representation of the median, quartiles and variables which might be considered outliers.

III. Results and Analysis

We apply the descriptive statistics on each of the 13 metrics to the ten projects. We present the results according to the internal component, interfaces and application.

A. Component Internal Metrics

1) Size Metrics: Table VI gives measures of metrics related to the size of components. Concretely, we want to highlight the management aspect of the component.

Table VI: Descriptive statistics for component's size metrics (case of Equinox)

Metric	Mean	Median	std. Dev.	skewness	Kurtosis
NP	3.1	2.5	2.43	1.54	2.29
Na	3.96	1	8.60	5	28.49
Nc	26	15	34.74	3.31	14.66

The important difference between the median and the mean, in the case of Na and Nc, has a sense for the characteristics of a distribution. The mean is affected by the outliers, thus making its value higher than the median. For these same two metrics the distribution is skewed strongly to the left with a very high and stronger peak, with the higher standard deviation values. These extremely high values can be explained by their heavy tail. There are many components with very few entities having either few abstract classes and concrete classes. Table VI provides an additional information on the package within component. For the NP metric, the mean is so close to the median and both skewness and kurtosis values are low, which leads us to conclude that the mean is significant and the NP in the majority of the components of Equinox application is close to the value of the mean. We presented the results for one application (Equinox), but this result is similar for other applications. In the following, we find this similarity through a graphical representation of the distribution of these metrics on these applications.

- Similarity between the applications

Figure 1 correspond to the size metrics distributions: The asymmetry of the distributions of Na and Nc (respectively

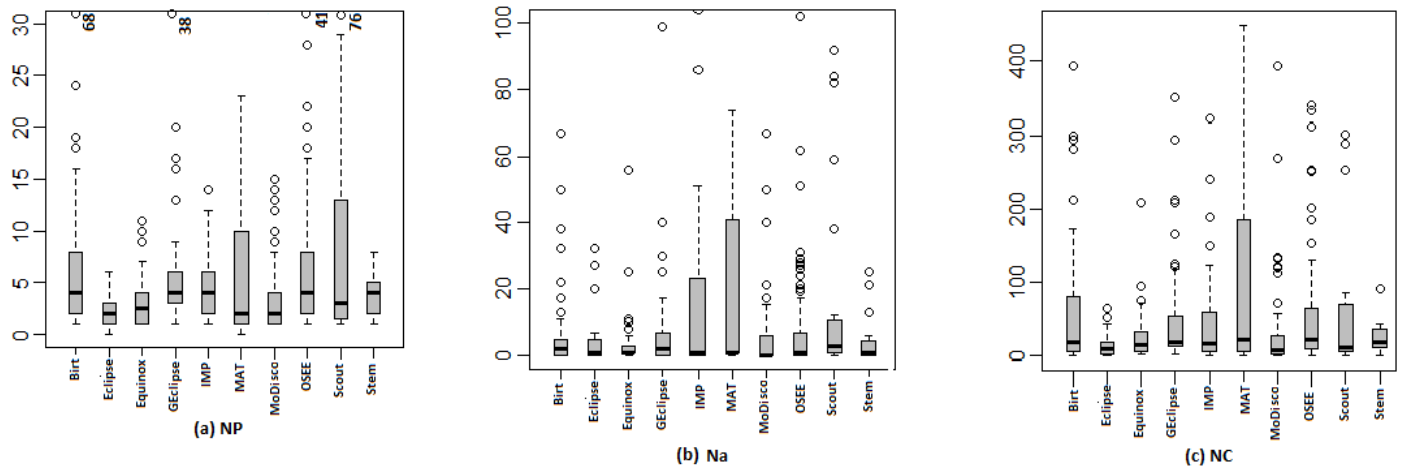

Figure 1: Component's Size Metrics Distribution

(b) and (c)) is very important for the majority of applications: values are strongly spread towards large values. The lower values are highly concentrated compared to the rest of the distribution.

Overall, for the NP metric, the median value of the metric is approximately the same for different applications. The medians are near to the value of three packages per component. So, the distribution of packages across components is very similar in the different applications whatever their sizes.

2) Internal Dependency Metrics: Table VII summarizes dependencies metrics within components of an application (Equinox). It shows how classes and packages are connected within a component.

Table VII: Descriptive statistics for component's internal dependency metrics (case of Equinox)

Metric	Mean	Median	std. Dev.	skewness	Kurtosis
LTD	49.20	17.5	119.55	5.73	36.43
PDC	0.86	0	1.48	2.62	9.90
ACD	4.06	2.56	6.68	6.04	39.93
RC	1.25	1.04	0.88	1.13	1.11

The relationship between classes in a component seems to be quite complex for some components. Indeed, the variation for LTD is very large. However, the interpretation of central tendency measures (mean and median) becomes unreliable in presence of highly skewed distributions.

Over the half of the components do not hold packages forming cycles (median equals 0). But, the PDC mean is 0.86 which confirms that there are components holding packages forming cycles. More precisely, there are few components which hold more than one cycle (standard deviation is 1.48).

RC have a low standard deviation equal to 0.88 with skewness and kurtosis near to 1. So, for RC the median is significant and most components have values around the median. In theory, the cohesion of a component should be high which means the contained classes should be strongly related. Given the importance of the RC metric, practitioners have defined thresholds. For example, for object-oriented paradigm (which is the paradigm used inside the component), the *NDepend*[1] tool strongly recommends that the value of this metric should be between 1.5 and 4. According to this recommendation, we can conclude that Equinox components are less cohesive.

- Similarity between the applications

Through Figure 2 we study the similarity between the applications concerning metrics on internal dependencies.

The observations for LTD (a) are broadly the same as for MAT application, distribution is asymmetrical and the values are highly spread towards large values. The same finding for the metric ACD (c). We note also that the MAT application is different at the higher dispersion values. Boxplots of the RC (b) show a fairly symmetrical distribution and that confirm results show for Equinox concerning the cohesion of components. PDC (d) has a very strong spread to higher values with the absence of the cycle for 50 % of individuals in the most of the applications and by against the presence of outliers.

B. Interface Metrics

Table VIII provides additional information on the package within component. ExpP and Abs have a skewness nearest to

Table VIII: Descriptive statistics for Interface metrics (case of Equinox)

Metric	Mean	Median	std. Dev.	skewness	Kurtosis
ExpP	2.52	2.0	2.03	1.17	1.16
Abs	0.167	0.125	0.148	1.05	0.36

a normal distribution. Since the mean and median are very close and both are near the peak, then the measure of central

[1]Metrics definitions and recommendations implemented in the NDepend tool (http:www.ndepend.com).

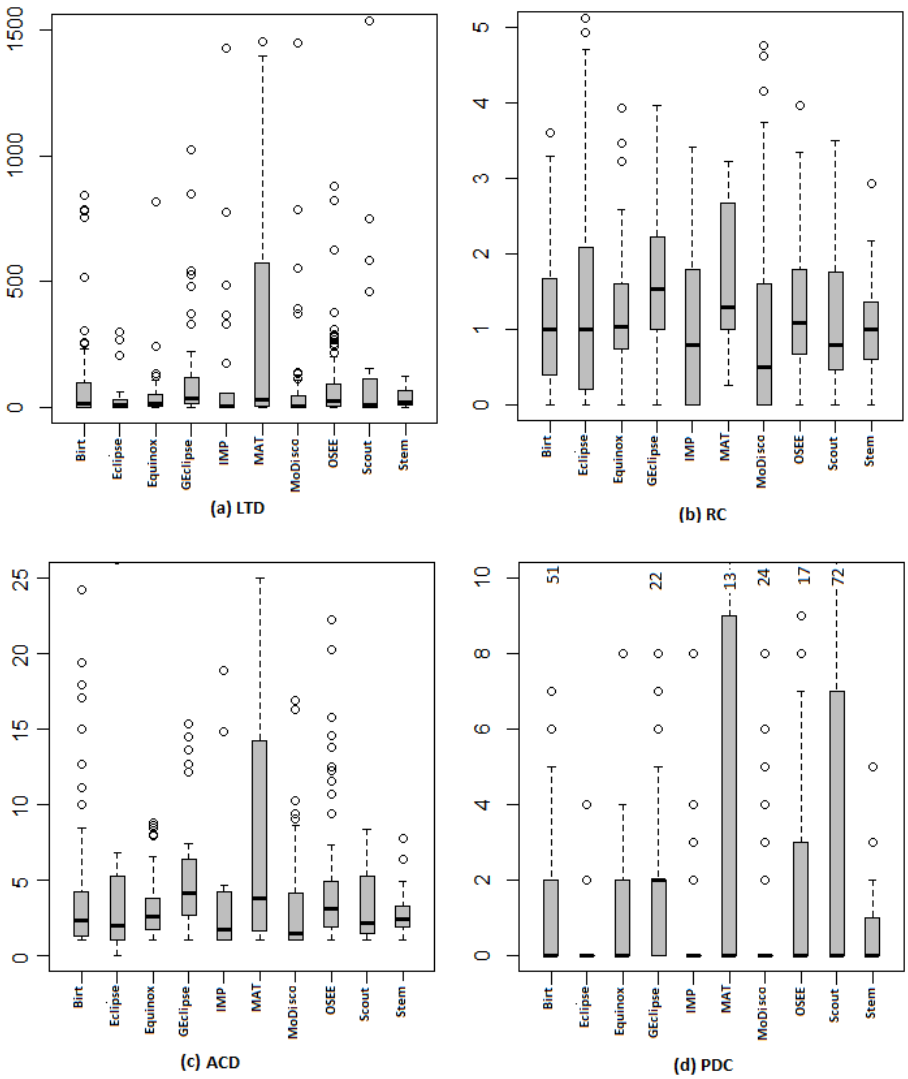

Figure 2: Component's Internal Dependencies Metrics Distribution

location is significant. This means that the median represents correctly the set of components from this application.

- Similarity between the applications

We can notice the asymmetry of the ExpP distributions (a): values are strongly spread towards low values for the majority of applications, by against a symmetrical distribution of abstractness (b) and the value of the median is very similar and so low for all applications.

C. Application Metrics

The final set of measurements concerns external dependencies. So, table IX shows the number of coupling between classes belonging to different components (RTD) and between components (CA, CE and Dep) for the case of Equinox. We can easily observe that the values of coupling between components (CA and CE) are low (median equals 1 for CE and 2 for CA). Indeed, over the half of the components do not use more than two components and are not used by more than one other component. However, the variation is quite high for both metrics mentioned above, especially for CE (the standard deviation is equal to 5.26). We can notice that the distribution for CE has extremely high values for skewness and kurtosis. Therefore, there are many components that are not used. However, there are few components that are very coupled with the others.

Table IX: Descriptive Statistics for Application Metrics (case of Equinox)

Metric	Mean	Median	std. Dev.	skewness	Kurtosis
RTD	109.8	61.5	115.76	1.33	0.83
CA	2.52	2	2.44	1.23	1.80
CE	2.54	1	5.26	3.16	9.71
Dep	4.96	4	4.34	1.67	3.14

For the RTD metric the values of skewness and kurtosis

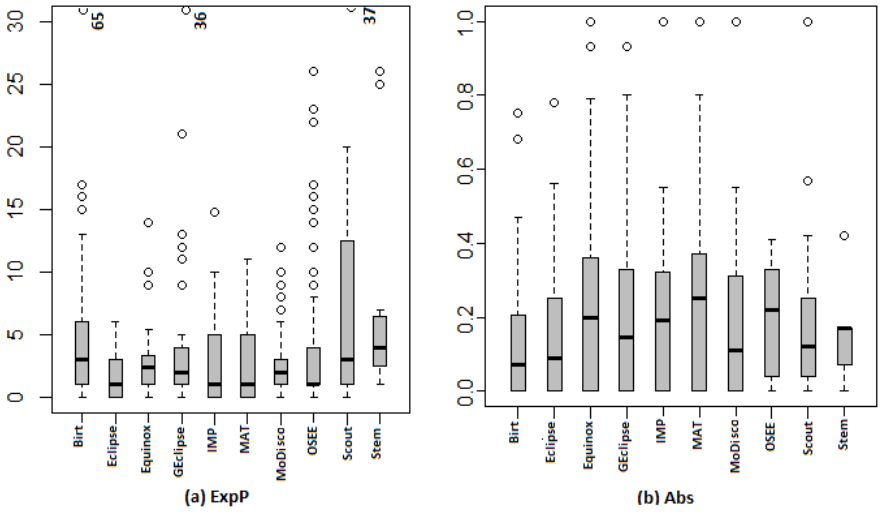

Figure 3: Interface Metrics Distribution

approach to 0 and this means that the value of median is very close to the peak and so, the value of central tendency of RTD metric is significant.

- Similarity between the applications

In figure 4 we can see the distribution of CA and CE respectively (a) and (b). We can show that there are many extreme values for CE. For example, a single component in Equinox application was used by 15 other components. Therefore, this component have too much responsibility and its changeability raises many problems. For the RTD (d), the lower values are highly concentrated compared to the rest of the distribution for the majority of applications.

D. Significance of used metrics

The majority of metrics of class granularity (Na and Nc (component size metrics), LTD and ACD metrics (for component internal dependencies) and RTD (metric for the application)) have similar distributions: an asymmetrical distribution spread by the higher values, a strong condensation to very low values and many outliers. Thus, these metrics taken alone seem not significant, but when we combine some of them they become significant. This is the case of LTD and Nc. Indeed, the most significant metric for the class granularity is the standard metric Rc which is a combination of LTD and Nc.

This granularity allowed us, in this preliminary study, to have a vision on the practices of developers in reality. But remains difficult to interpret in the daily of developers, with the exception of the metric Rc, and can't consider that the class granularity is interesting for the component paradigm. We admit that a good adaptation of these metrics makes them more interesting.

By opposite, metrics of package granularity are significant. Indeed, the study of the distribution of the metrics NP, ExpP, Abs and PDC shows that they are significant. Indeed, the results for these metrics have a symmetrical distribution despite of the existence of some outliers. Thus, the metric of package granularity seem interesting to study the quality of the components.

Similarly, metrics of component granularity are also significant: CA, CE and Dep metrics have distributions that allow us to see the importance of a component and its degree of dependencies with the other components and consequently, it helps to improve reusability.

IV. Discussion

We discuss the results of our experimentation according to two points of view: the component developer and the architect who reuses developed components for building applications. The former is concerned by component the internal and the interface metrics while the latter is concerned by the application metrics.

A. Component Developer Point of View

Figure 1 shows that few number of components hold a big value of Nc and Na while the other components are very lightweight. So, the former group of components seems to be the core of the application. But their large size implies low maintainability. This often happens when legacy application is restructured into component-based application. Indeed, developers find difficulties in adjusting the granularity of core components of the application.

If we examine the distribution of the PDC metric (boxplot (d) of Figure 2) we can notice that all applications contain

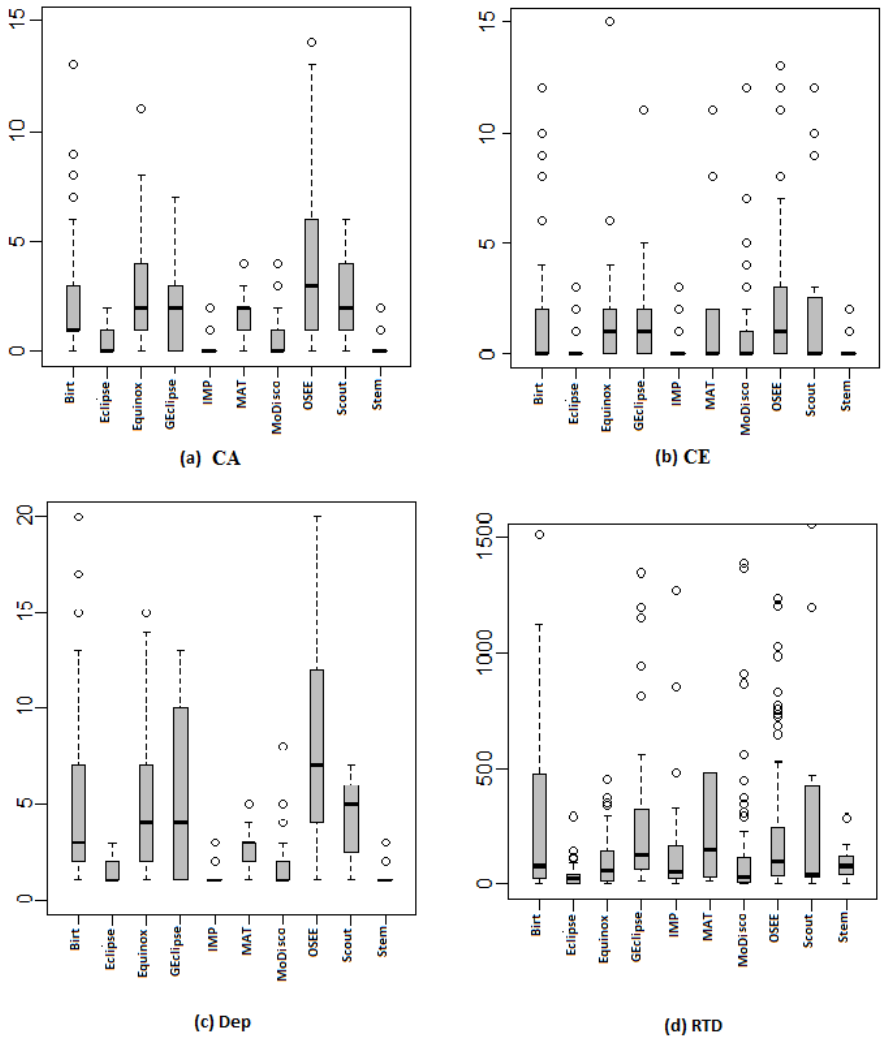

Figure 4: Application Metrics Distribution

components having cycles. Therefore, one of the two most important criteria to ensure the reusability of a component wasn't observed. But, with a median equal to 0 for most applications (9/10) means that more than half of their components have no cycles.

The general definition of a component specifies that the latter must provide a unique functionality (a restricted number of exported methods). In practice, it is unrealistic to expect that all components export a single functionality. This definition should be respected by the majority of the components of an application. But, as shown in Figure 3 there is a lot of extreme values for the most of applications (eg. 65 exported packages for the same single component in Birt Application). Therefore, developers should pay attention to packages that are intended to be used by other components. Exporting a large number of packages may mean that the component has multiple functionalities. It would be better to decompose it into several components [12].

In addition, we can see that the abstractness is very similar for all exported packages in all applications. In fact, the most are ranged from 0 to 0.3 which means that they are more concrete than abstract. However, to avoid the interdependence between components (for good reusability) exported packages must contain only definitions (not concrete classes).

From the remarks noticed above, it is clear that the developers of the selected component-based applications have ignored the inside quality of components. They do not respect rules such as the component must be a cohesive entity and without cycles formed by its constituents [17].

B. Architect Point of View

From Figure 4 we can notice that the values of coupling between components (CA and CE) are low for all applications. Furthermore, by analyzing the median number of packages exported by component (ExpP), the average number of classes per component (Nc) and the RTD metric, we can conclude that the number of method calls between exported packages remains relatively high (about 100 calls per package). This demonstrates a good application design because despite a weak coupling between the components, existing connections have a

good flow of communication. This means that the components were well used despite their poor internal design.

V. Threats to validity

This empirical study has several threats that may restrict the generality and limit the interpretation of our results. So, in the following we discuss the internal and external validity for the selected systems and the selected metrics.

A. Internal Validity

The measurements of chosen metrics were performed with a well-known tool (SonarGRaph) and therefore for which there is a real policy of correctness and bug tracking. It is therefore reasonable to assume that the measures are consistent with the definitions of metrics. That is to say that a different tool implanting the same definitions of metrics obtain the same values. However, for some metrics, among those we selected, there are some variations in their definitions and implementations. Thus, we can not ensure that the use of another tool for the calculation of these metrics lead to the same results that we have obtained.

In addition, some metrics have been achieved thanks to the structuring of applications required by the OSGi framework. This is the case for example for the Abs metric (Abstractness only for exported packages). Indeed, the framework clearly identifies the exported packages with the manifest file. But the identification of component interfaces, through their source code, may be more dificult for another framework.

B. External validity

Our study focuses on 10 OSGi applications. These applications were chosen to represent different sizes, different development teams and different application areas. We have some confidence in the conclusions on the existence of common features regardless of the choice of applications. However all applications are developed with Java. Thus, we can not guarantee the same result for another object-oriented language (eg C + +). Indeed, the structural aspect of Java can facilitate the identification of common properties for components, as it can hide others.

In addition, this study is devoted exclusively to the OSGi framework. It has features that differentiates it from other framework based also on object-oriented languages. For example, the concepts of container and the possibilities of reflexivity in Fractal framework or the additional classes required by EJB framework are likely to change the way a component is structured. This can obviously cause a significant change in the ranges required for certain metrics. This study is therefore probably not generalizable to frameworks defining the contours of a component and their relationships very different from that of OSGi.

VI. Related Work

Quality for component-based applications is a relatively new field of study. Several work have appeared in the last few years in this field. But, it is clear that there is no consensus on properties for this kind of software. These work concerns either metrics for component-based applications or quality models for component-based applications.

A. Metrics

We can distinguish different types of study that relies on metrics dedicated to individual component and their assembly in general. Some studies focus on metrics calculating only the external properties of a component or a component-based application, not taking into account component internal properties.

For instance, Washizaki et al. [1] measured the reusability of black-box components.They defined six metrics for measuring understandability, adaptability, portability and reusability of a JavaBeans component. Similarly, Wei et al. [5] proposed a set of metrics to evaluate the architecture of a component-based software system. This set of metrics was defined using the Component Assembly Graph. These studies disregard the internal code. However, some studies such as [23], [4] have emphasized the importance of the internal code for a component. They consider that internal design can not be simply ignored since some internal attributes of a component may provide an indirect measurement of its external characteristics.

Thus, some work propose to take into account the internal structure of components [6], [7], [24]. For instance, Cho et al. [6] provided new metrics for measuring the complexity, customizability and reusability of software components. One of these metrics is the CPC metric, which is the aggregation of classes, abstract classes, interfaces and complexity classes and methods. Likewise, Choi et al. [7] proposed new component-based cohesion and coupling metrics applying strength of dependency between classes.

Nevertheless, some studies are interested in the quality of the interfaces of software components whilst requiring access to the implementation and hence, suggested some metrics to measure the properties of the component interface [25], [26].

All these different types of study are useful for developers, since they must analyze not only the quality of the internal (hidden) structure of a component but also the quality of its visible parts.

Right now, there is no evidence that any of the proposed metrics can predict the component quality effectively [8]. So, these proposed metrics need systematic studies to determine their validity and relevance with real information.

B. Quality Models

Several quality models specific to software components and to component-based applications have been defined. Most of them are based on the ISO 9126 quality model, with some changes to make them suitable to the component-oriented paradigm. In this work, the authors often needed to define new metrics in order to build their model [2], [27], [28], [1]. But it was pointed out in [29], any of them has become accepted as "standard" and most of them have not yet been validated.

Our approach is not based on a specific quality model. We just selected a suite of metrics that can provide guidance

to developers on the code's quality of their component or to architects on the quality of their application. Thus, our proposal can be complementary to the work on quality models.

VII. Conclusion

The primary objective of the study presented in this paper is to show that it is possible to use metrics from object-oriented paradigm in order to measure some characteristics of component-oriented applications built upon object-oriented technology. We have shown that for component-oriented applications built with the OSGi framework. We proposed metrics for each component-oriented application development actors (component developer and application architect). But we believe that this study is substantially reproducible for other frameworks for building component-oriented applications. Indeed, the metrics which measure has been facilitated by the OSGi framework, their measure within other frameworks remains feasible.

By separating the metrics related to the construction of components from these metrics related to the construction of applications we highlighted another aspect of the practice of CBSE: the architects (the component users) seem to know and observe the component-oriented concepts, while component developers seem far away. But the context for the latter was not easy. Indeed, the use of object oriented approach has obvious consequences. Often to construct a component, the developer reuses existing classes with their packages. Thus, she/he undergoes some internal measures without being able to influence them. The problem is that a component can be considered as a package, but the mode of structuring a component is quite different from that of a package.

We believe we have initiated the construction of a repository of well-validated metrics from object-oriented paradigm that can be used in case of component-oriented paradigm when it is implemented upon object-oriented technology. Thus, further studies can complement this repository. Our future work will involve the use of the metrics identified by this work to propose some prediction models for component properties such as maintainability and reusability.

References

[1] H. Washizaki, H. Yamamoto, and Y. Fukazawa, "A metrics suite for measuring reusability of software components," in *IEEE METRICS*, 2003, pp. 211–.

[2] M. F. Bertoa and A. Vallecillo, "Quality attributes for cots components," 2002.

[3] S. Sedigh-Ali, A. Ghafoor, and R. A. Paul, "Software engineering metrics for cots-based systems," *Computer*, vol. 34, no. 5, pp. 44–50, 2001.

[4] K. K. Chahal and H. Singh, "A metrics based approach to evaluate design of software components," in *ICGSE*, 2008, pp. 269–272.

[5] G. Wei, X. Zhong-Wei, and X. Ren-Zuo, "Metrics of graph abstraction for component-based software architecture," in *Computer Science and Information Engineering, 2009 WRI World Congress on*, vol. 7, 31 2009-April 2, pp. 518–522.

[6] E. S. Cho, M. S. Kim, and S. D. Kim, "Component metrics to measure component quality," in *APSEC*, 2001, pp. 419–426.

[7] M. Choi, I. J. Kim, J. Hong, and J. Kim, "Component-based metrics applying the strength of dependency between classes," in *SAC*, 2009, pp. 530–536.

[8] M. Abdellatief, A. B. M. Sultan, A. A. A. Ghani, and M. A. Jabar, "A mapping study to investigate component-based software system metrics," *Journal of Systems and Software*, vol. 86, no. 3, pp. 587–603, 2013.

[9] S. Allier, H. A. Sahraoui, S. Sadou, and S. Vaucher, "Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces," in *CBSE*, 2010, pp. 216–231.

[10] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. Software Eng.*, vol. 33, no. 10, pp. 709–724, 2007.

[11] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, "A classification framework for software component models," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 593–615, 2011.

[12] S. M. Richard S. Hall, Karl Pauls and D. Savage, "Osgi in action creating modular applications in java," 2011.

[13] S. Chardigny, A. Seriai, D. Tamzalit, and M. Oussalah, "Quality-driven extraction of a component-based architecture from an object-oriented system," in *CSMR*, 2008, pp. 269–273.

[14] S. D. Kim and S. H. Chang, "A systematic method to identify software components," in *APSEC*, 2004, pp. 538–545.

[15] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *QUATIC*, 2007, pp. 30–39.

[16] S. Jungmayr, "Design for testability," in *In Proceedings of CONQUEST 2002*, 2002, pp. 57–64.

[17] J. Lakos, "Large-scale C++ software design." Addison-Wesley, 1996.

[18] R. C. Martin, "Agile software development: principles, patterns, and practices," 2003.

[19] C. Larman, "Applying uml and patterns," 2002.

[20] M. Vieira and D. J. Richardson, "Analyzing dependencies in large component based systems," in *Proceedings of the 17th IEEE International Conference on Automated Software Engineering, Edinburgh, UK*, 2002.

[21] B. Zeiss, D. Vega, I. Schieferdecker, H. Neukirchen, and J. Grabowski, "Applying the iso 9126 quality model to test specifications - exemplified for ttcn-3 test specifications," in *Software Engineering*, 2007, pp. 231–244.

[22] Y. Dodge, *The Oxford Dictionary of Statistical Terms , Oxford*, 2003.

[23] B. Meyer, "The grand challenge of trusted components," in *ICSE*, 2003, pp. 660–667.

[24] E. Lee, B. Lee, W. Shin, and C. Wu, "A reengineering process for migrating from an object-oriented legacy system to a component-based system," in *COMPSAC*, 2003, pp. 336–341.

[25] O. P. Rotaru and M. Dobre, "Reusability metrics for software components," in *AICCSA*, 2005, p. 24.

[26] M. A. S. Boxall and S. Araban, "Interface metrics for reusability analysis of components," in *Australian Software Engineering Conference*, 2004, pp. 40–51.

[27] A. Alvaro, E. Almeida, and S. Meira, "Towards a software component quality model," in *Submitted to the 5th International Conference on Quality Software*, 2005.

[28] A. Rawashdeh and B. Matalkah, "A new software quality model for evaluating cots components," *Journal of Computer Science*, vol. 2, no. 4, pp. 373–381, 2006.

[29] S. Kalaimagal and R. Srinivasan, "A retrospective on software component quality models," *ACM SIGSOFT Software Engineering Notes*, vol. 33, no. 6, pp. 1–10, 2008.